CSC A48 Winter 2019 Term Test Study Guide

Section 0: Introduction to C and Basic Data Types

C is an **imperative** language. There is no interactive mode in C. All programs start at a function called `main()`. Program can include and use code from **libraries**, similar to Python modules. Here is an example of the most famous program in C.

```c
// Allows you to use input and output command such as printf
#include <stdio.h>

int main()
{
  // end your print statement with a newline character
  printf("HELLO WORLD\n");
  // your main function must return 0 if successful
  return 0;
}
```

Compiling, Running and Makefiles

Data Types

The fundamental data types supported by C are: 1. `int` which is an integer number, 2. `float/double`, a floating point number 3. `char` which is one character and 4. `void` which represents no data type attached like `None` in Python.

```c
#include <stdio.h>

int main()
{
  float my_float = 10.0;
  int my_int = 3;
  char my_char = 'a';
  printf("My variables are: %f, %d, %c \n", my_float, my_int, my_char);
  return 0;
}
```

Arrays

An indexable collection of data. Recall that you must declare: 1. the size of the array 2. what type the items are

Unlike variables in Python, you can't have items of different types in an array in C. i.e In Python, you can have a list `L = [0, 'hi', [2, 1, 1], {'s':3}]` whereas in C, every item in array must be the same type.

```c
// array of ints
int numbers[5];  // doing int numbers[5] = {} initializes all values to 0
// array of chars (string)
char letters[5]; // doing int letters[5] = {} initiliazes all values to ''
// initializes a 2d array
int numbers3d[2][3]; // think "Arrays of Arrays" or "LISTS of LISTS in Python"

// you can set the values of your array directly.
int nums[5] = {1, 2, 3, 4, 5};
// same for a 2d array
int matrix[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
// Recall that arrays a pointer to a single element. You are printing the memory address.
printf("%d\n", nums);

printf("%d\n", *nums); // You can only print one item at a time, this prints the first one.
printf("%d\n", nums[3]); // print an item at an index
printf("%d\n", matrix[1][2]); // should give 6
nums[3] = 2; // change value at an index
printf("%d\n", nums[3]); // used to be 4 now should be 2
```

```
-9999999999999 // or whatever ugly numbers your program wants to use
1
4
6
2
```

You can only access the items in an array one by one, so you will have to print them by looping through them. This is unlike python where you can just `print(List)` and the entire List will print for you.

Iteration Over an Array

```c
int nums[5] = {1, 2, 3, 4, 5};
// you can get array length by dividing total size by size of the type used
int length = sizeof(nums)/sizeof(int); // in this case total size = 20, int size = 4

// both of the loops below do the same thing
// indexed for loop
int i;
for (i = 0; i < length; i++) // Python Equivalent: for i in range(0, length, 1):
{
  printf("%d\n", nums[i]);
}

// while loop
int i = 0;
```

```c
while (i < length)
{
  printf("%d\n", nums[i]);
  i ++;
}
```

```
BOTH OUTPUT:
01234
```

Notice, we didn't do `for (int i = 0; ...)` which you may be familiar with doing in Java or C# or some C based related language. This doesn't work unless you add `-std=c99` (c version 99) in gcc. You should initialize the i just to be safe. C does not have a "for-each" (equivalent for `for item in items`) implementation, but you can accomplish what you need with the index for loop.

Section 1: String Operations

Strings in C

Recall that strings in C are not primitive. Strings are an **array of characters**, which is terminated with a **null character** `\0`. You must end your strings with the null character or else you won't be able to perform operations like printf.

`string.h` library

strcpy and strncpy

```c
char a[10], b[10];
strcpy(a, "Hello");
printf("%s\n", a);
printf("Copying first four letters of a to b\n");
strncpy(b, a, 4);
printf("%s\n", b);
```

```
Hello
Copying first four letters of a to b
Hell
```

strcat and strncat

```c
char a[10], b[10];
strcpy(a, "Hello");
strcpy(b, "World");
strcat(a, b);
printf("%s\n", a);
printf("concatenating first 7 letters of a to b\n");
strncat(b, a, 7);
printf("%s\n", b);
```

```
HelloWorld
concatenating first 7 letters of a to b
WorldHelloWo
```

strcmp and strncmp

```c
char a[10], b[10];
strcpy(a, "Word");
strcpy(b, "World");
// if the words are equal, strcmp will return 0
if (strcmp(a, b) == 0)
{
    printf("%s and %s are equal\n", a, b);
}
else
{
    printf("%s and %s are not equal\n", a, b);
}
int n = 3;
if (strncmp(a, b, n) == 0)
{
    printf("%s and %s are equal to %d bytes\n", a, b, n);
}
else
{
    printf("%s and %s are not equal\n", a, b, n);
}
```

```
Word and World are not equal
Word and World are equal to 3 bytes
```

Section 2: Pointers

Any programming language at its very heart is about storing and moving information around. In fact, to your computer, there is no difference between an `int` or `char` or even `char*` etc., they are simply a sequence of bits that are passed around in your machine.

It is the human component and only this component that makes the decision that an `int` expresseses our idea of quantity or number and `char` expresses our idea of characters. If you were to "ask" your computer what is stored at a particular memory address, the response will always be "I don't know. Who cares?". The first byte at this location could be interpreted as a `char` or the first 4 or 8 bytes at the same location (depending on your machine) could be interpreted as an `int`, `char*` or any other data type.

We can let the compiler know we've decided to switch our interpretation of the data we stored in a variable. It's called casting

```
int main(int argc, char ** argv) {

    char mem[] = "Hello World";
    // creating a int pointer to the array allows us to use the same
    // data, but interpreted differently.
    int * p = (int*) &(mem[0]);

    printf("Size of Integer: %d bytes\n", sizeof(int));
    printf("%d", *p);
}
```

```
$ gcc casting.c
$ ./a.out
Size of Integer: 4 bytes
1819043144
```

> Note: Since the size of integer is only 4 bytes, not all of "Hello World" has been translated to a integer value. only the first 4 bytes.

The point being, **memory isn't specifically "formatted" to store your requested data type**. An `int` is stored in the same way as a `char`,`char*`, `double` etc. - a sequence of bytes. A data type simply defines how many bytes we should read starting from a location.

Pointers in C are like any other datatype. When declaring a pointer, your computer simply allocates 8 bytes in memory to store a sequence of 8 bytes. It is us who decided to first interpret these 8 bytes as a number then as a memory address.

Whenever you pass in a variable to a function in C, it is always copied. If its an `int`, the bytes of size `sizeof(int)` are copied. If its a `char*` the bytes of size `sizeof(char*)` are copied. These copied values exist as local variables inside your function, specifically parameters, and go out of scope when the function returns. The same logic is true for returning a value from a function, the value is copied and sent back to the caller.

Common Mistakes and Bugs

Uninitialized Pointer - Dereferenced before assignment

```
int main(int argc, char ** argv) {

    int * p;

    *p = 3; // BAD! p doesn't point to a valid location in memory.
```

```
            // Attempting to write will trigger SIGSEGV
}
```

Dangling Pointer - Access

```
int main(int argc, char ** argv) {

    char * p = malloc(20); // allocate 20 bytes
    strcpy(p, "hello");
    ...
    free(p);
    strcat(p, "world"); // BAD! memory was already freed
                        // Attempting to access will trigger SIGSEGV
}
```

Memory Leak

```
int main(int argc, char ** argv) {

    char * p = malloc(20); // allocate 20 bytes
    strcpy(p, "hello");

    char a[] = "Cake";

    p = &(a[0]); // BAD! p was pointing to memory allocated on heap
                 // reassignment of p now removes all references to that
                 // memory location. cannot free.
}
```

Difference between Pointers and Arrays

Arrays and Pointers are similar in how they access memory. But their key difference lies in how they act and the operations you can perform on the two.

```
int arr[3]; // allocates space on the stack for 3 integers

arr[0] = 5; // assigns the first element in the array to 5

printf("Indexed Value: %d\n", arr[0]);
printf("Dereferenced Value: %d\n", *arr);
```

The two expressions will both yield the first value of the array.

```
$ gcc array.c
$ ./a.out
Indexed Value: 5
Dereferenced Value: 5
```

However, unlike declaring a `int *arr`, we cannot reassign `arr` to another value. In other words, arrays behave very much like a pointer, but is NOT a pointer.

```
int p[10];
int d = 9;
p = &d; // illegal, gcc will complain.
```

Array Pointer Decay

When an array is passed into a function, it is said to decay into a pointer to the first element.

```
void f(int * p) {

    p[0] = 10; // legal!

    int d = 5;
    p = &d; // reassignment is also legal :)
}

int main(int argc, char ** argv) {
    int r[10];

    f(r);
}
```

If you understand that all parameters passed into a function are copied, this makes perfect sense! Calling `f(r)` copies the address of the first element in the array `r` and passes it into `f()`.

Dereferencing a pointer and overwriting the value

We can also dereference a pointer and overwrite the value at the location it's pointing to. This allows us to change the value of variables inside a function implicitly or simply later in the code.

```
void lie(int * age) {
    *age = 19;
}

int age = 10;
// here age == 10
lie(&age);
// here age == 19
if (age >= 19) {
    printf("You are allowed to drink alcohol")
} else {
    printf("You are too young!")
}
```

Section 3: Structs

Structs in C allow us to define a new data type similar to the `class` keyword in Python. However, C being a much lower level language than Python, does not offer a lot of the luxuries you have enjoyed in Python. We cannot cannot create methods nor specify access modifiers in our struct.

A `struct` is essentially a contiguous chunk of virtual memory used to store pieces of data. The number of members in the struct and their respective data types determine the size of the `struct`. It is important note, while the entire struct is stored contiguously, the members of the struct are not neccessarily contiguous (i.e there is sometimes padding between members). The reason for this is out of scope of this course, you will learn more about this in CSCC69. For now, it is important to remember, you should ALWAYS determine the size of a `struct` by using the `sizeof()` operator and not adding up the size of its members.

```
struct Person {

    char name[MAX_NAME_LEN];

    int age;

    int birthyear;
    int birthmonth;
    int birthdate;

    double weight;
    double height;
}

int main(int argc, char ** argv) {

    // we can allocate a struct on the stack
    struct Person newPerson;
    // we can set properties
    strcpy(newPerson.name, "David");
    newPerson.age = 20;
    newPerson.birthyear = 1998;
    newPerson.birthmonth = 9;
    newPerson.birthdate = 27;

    // like wise, we can also allocate a struct Person on the heap
    struct Person *second_person = malloc(sizeof(struct Person));
    // to access and assign the member values, use the -> operator
    // since we have a struct Person pointer
    second_person->age = 10;
    ...
```

```
}
```

It is important to note that if one of the `struct` members is a pointer, pointing to memory allocated on the heap, you MUST FIRST free the pointer member first before you free the struct.

Section 4: Linked Lists

You will use linked lists a lot during your upper year courses. It is a very fundamental data structure and mastery in manipulating linked lists is vital.

A linked list comes with some basic operations as you have seen in lecture. While even a lackluster CS student could see how linked lists could be created in Python with a class containing a `next` property. The pointers in C actually give us a whole new level of control. The ability of being able to create a pointer and dereference it somewhere later in your code is a very powerful tool.

Let's review some of the basic operations and how this dereference operation will fit in:

Let's first define our Node `struct`

```
typedef struct __node__ Node; // don't worry about this, this is called
                              // prototyping. It's so we can use the
                              // typedef keyword 'Node' in our Node struct
                              // before we've actually defined it.

typedef struct __node__ {
    int v;
    Node * next
} Node;
```

`INSERT(head,k)`

Note the double pointer to head allows us to change the head node pointer by dereferencing.

```
/**
 * Inserts the Node k into the beginning linked list head.
 */
void insert (Node** head, Node* k) {

    if (!head) return;
    k->next = *head;
    *head = k;
}
```

```
/**
 * Inserts the Node k into the end of linked list head.
 */
void insert (Node** head, Node* k) {

    if (!head) return;

    Node * curr = *head;

    // if this is the first node in the list, overwrite head.
    if (!curr) *head = k; return;

    // find last node in list
    while (curr->next != NULL) curr = curr->next;
    // set the next to our new node.
    curr->next = k;
}
```

DELETE(head,v)

Delete is slightly more complicated as we have to consider the edge cases where the node is the head of the list and the end of the list

```
/**
 * Deletes Node with value v from linked list head
 * Returns 0 if node with value v is not found, 1 otherwise
 */
int delete (Node** head, int v) {

    if (!head) return 0;

    Node * curr = *head;
    Node * prev = NULL;

    // look for the node with value v.
    while (curr->v != v) {
        // keep track of previous node, we will need it later
        prev = curr;
        curr = curr->next;
        // if we reach end of list prematurely, just return
        if (!curr) return 0;
    }

    // it helps to consider the exit condition of this loop
    // exit condition: negation of while loop condition
    // -> curr->v == v
```

```
    // this means the loop will exit with curr->v == v

    if (!prev) { // if the node we want to delete is the head node.

        *head = curr->next; // overwrite the head node pointer to the next one.

    } else { // otherwise it occurs at the end of the list or the middle

        // if it's in the middle, this sets the previous node pointer
        // to the one after curr (curr being the one we want to delete).
        // if it's at the end, curr->next == NULL, hence still correct
        // as the prev node will now have prev->next = NULL.
        // (i.e the new end  of the list)
        prev->next = curr->next;
    }
    // free up the memory for curr (the deleted node)
    free(curr)
    return 1;
}
```